Hi-Res: Precise Exploit Detection using Object-Granular Memory Monitoring

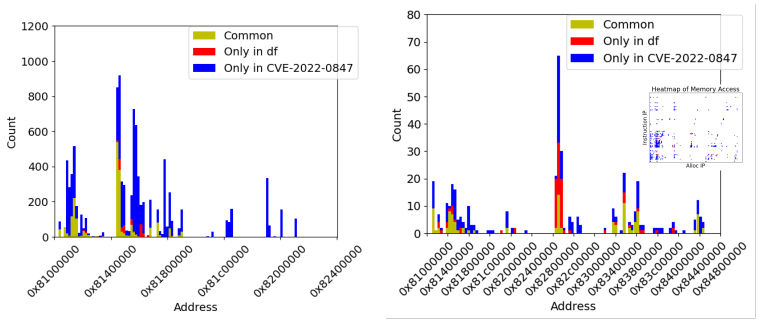
Ziyang Yang, Saumya Solanki, Scott Rixner, and Nathan Dautenhahn

Abstract. Despite numerous methods for identifying, preventing, and protecting against kernel-level exploits, attacks persist. One of the key challenges is the prevalence of *weird machines*—unintended computational artifacts that attackers dynamically stitch together from unmonitored low-level operations. This paper presents Hi-Res, a programmable detection framework that systematically lifts high-level exploit behaviors from their low-level memory operations. Unlike traditional methods that rely on expert-driven, hand-crafted monitors, Hi-Res automatically generates a unique fingerprint of kernel execution given a specific input and execution contexts. Hi-Res projects memory traces into a high-resolution hyperplane, where behavioral fingerprints are constructed from observed access patterns. Using this representation, Hi-Res, is able to explore the hypothesis that low-level program traces exhibit locality properties that are distinct, context-sensitive memory access patterns unique to specific workloads. This locality coupled with the concrete Hi-Res representation enables the empirical modeling of working sets without prior knowledge of program semantics. By analyzing specific dynamic context tuples—such as system call, access-from location, allocation contexts, and call stacks—we demonstrate that these fingerprints reliably differentiate between normal and exploit behaviors. Our results confirm that locality serves as a robust signal for precise exploit detection, establishing Hi-Res as a general, data-driven framework for dynamic security monitoring.

1 Introduction

The Linux kernel, with its extensive codebase contributed by thousands, represents a critical component in modern computing environments. Despite its robustness, the kernel's monolithic architecture and shared protection domain expose it to significant security vulnerabilities. Current kernel-hardening efforts, such as the Linux Self-Protection Project (KSPP), have made strides in fortifying the kernel against exploitation [2–4, 6]. Yet, these measures often offer only coarse-grained protection and lack the precision needed to counter sophisticated attacks that leverage multiple exploit techniques simultaneously [5, 11, 20, 21].

This gap in defense capabilities highlights the necessity for a more granular approach to exploit detection. Consider the challenge of detecting a sophisticated kernel-level exploit aimed at privilege escalation (as depicted in Figure 1), a common and dangerous threat in cybersecurity. Traditional detection systems might monitor for anomalies or signatures of known attacks, but sophisticated adversaries can evade these by employing novel techniques or combining multiple vulnerabilities [7, 8, 22–24]. For instance, an attacker might exploit a



(a) Feature: `access_ip` (b) Feature: `alloc_ip`

Fig. 1. Distribution of memory accesses for different features. `access_ip` stands for the address of the memory accessing instruction. `alloc_ip` stands for address of the `malloc()` instruction of the accessed object. Note that the memory accesses are also visualized as a mini heatmap shown on the right-hand side where the axis ranges correspond to the main figures.

previously unknown bug to manipulate kernel data structures subtly, bypassing coarse-grained monitors designed to catch more overt malicious activities. This limitation underscores the need for a detection mechanism that operates with finer granularity and higher accuracy.

Our research introduces a novel methodology and system, Hi-Res, that addresses this need by focusing on the unique behaviors exhibited by attacks at the object level. This approach is grounded in the hypothesis that malicious activities will manifest distinct patterns that can be detected through careful analysis of memory access behaviors and context. In essence, attack semantics are noticeably different at object granularity because objects represent types and types represent semantics. As captured by the object level access graph, attack behavior becomes visible without requiring any extensive supervised learning methods. In addition to the increased precision, we also tackle the challenging problem of efficiency for full object tracing by developing a live monitor swapping mechanism to record only suspicious process contexts.

To validate our approach, we conducted a comprehensive analysis of various workloads and exploits, examining their memory access patterns to inform the development of both signature-based and anomaly-based detectors. Our study extends beyond simple detection, exploring the impact of different fine-grained features on the accuracy and efficiency of these detectors. The culmination of our research is the creation of an in-kernel monitor that provides detailed insights into kernel-mode behaviors with minimal performance overhead, marking a significant advancement in the field of kernel exploit detection, and contributing:

– a new methodology and system, Hi-Res, for detecting kernel exploits through object-granular behavior monitoring,

– demonstrate the effectiveness of this approach through a comparative study of normal and exploit workloads (§5), and
– introduce an efficient in-kernel monitoring framework designed for real-time exploit detection (§4).

This work not only bridges the gap in current kernel protection mechanisms but also sets the stage for future advancements in securing critical systems against increasingly sophisticated threats.

2 Threat Model

We assume that the attacker starts by compromising an untrusted process so that she can execute arbitrary code in the untrusted process. To interact with the rest of the system, the attacker executes unprivileged instructions (e.g. to fabricate data) or goes through the software interface exported by the operating system abstract binary interface (ABI) and leverages knowledge about the internals of the kernel. We also assume that all the trusted processes are benign and free of vulnerabilities so they can not be exploited by the attacker via mechanisms like inter-process communication (IPC). Our work intends to study the detection of software-based exploits. Therefore, hardware-based attack techniques like Meltdown [14] are out of scope.

3 Motivation and Background

As we described in Section 1, our approach to exploit detection is based on the visibility of the kernel's fine-grained behaviors. Figure 1 shows the object-level memory access patterns for two workloads: CVE-2022-0847 exploit and df. The exploit is based on an arbitrary write enabled by the DirtyPipe vulnerability, while df is a utility showing file system information. From the view of existing coarse-grained defenses, they look the same: neither involves illegitimate user-space access or corruption of stack canaries. But in terms of memory access trace, those two are clearly different: the exploit not only modifies the (supposedly) read-only pages in the page cache, but also traverses code paths and touches objects for triggering the vulnerability, while the normal one does not. Moreover, normal workloads like df never touch `struct cred` and escalate their own privilege.

To gain visibility of the fine-grained behaviors that existing defenses lack, we build an in-kernel monitoring framework whose mechanisms are built on top of the techniques pioneered by Memorizer. Memorizer is the first of its kind to obtain a *complete* trace of all the object-level memory accesses from within the Linux kernel as far as we are aware [18]. To obtain such traces, one of the key innovations of Memorizer is the combination of source code instrumentation and compile-time instrumentation. It (1) instruments the Linux source code for key object lifecycle events such as allocation and free for all objects so that it can maintain an address-to-object mapping for each byte in the memory at runtime. (2) adds compile-time instrumentation for a complete instruction-level memory

access trace and leverages the address-to-object mapping to lift the trace to the object level on the fly.

3.1 Address-to-Object Mapping

The mapping is implemented as a three-level hash table that mimics the page tables for address translation, though its granularity is byte rather than page. Given a memory address, there is a unique path along the hash table hierarchy such that the leaf node contains a pointer of its metadata (serving also as an object identifier) that the address belongs to.

To maintain the mapping, Memorizer adds hooks to the "allocate" methods of the kernel so that each time an object is allocated, the entries corresponding to the bytes covered by the object are populated. In order to perform an address-to-object translation, one simply walks the hash table hierarchy from the top to the leaf node. The "free" methods of the allocators are also hooked so that the entries corresponding to the bytes covered by the object are cleared when the object is freed.

3.2 Object-level Memory Access Tracing

The standard object model [13, 16] is used by Memorizer to represent memory accesses. Each access can be denoted as a tuple `(subject, object)` where the subject is simply the address of the memory accessing instruction, and the object is an identifier in some form. Note that it is possible to enrich the definition of subject and objects with extra runtime states when the object allocation or memory access events happen, at the cost of additional memory overhead.

Memorizer generates a complete object-level memory access trace by adding compile-time instrumentation to the kernel for a complete instruction-level memory access trace and leveraging the address-to-object mapping described in the previous subsection to lift the trace to object level on the fly.

As a proof of concept, Memorizer simply hooks the Kernel Address Sanitizer (KASAN)'s runtime. KASAN [1] uses a GCC compiler pass that instruments memory accessing instructions. The following code snippet shows an example of how it works conceptually. For a read to the memory address saved in register `%rbx`, the compiler pass adds the call to `__hook_load8()` with the argument as `%rbx`. Note that `%rdi` is one of the conventional registers for passing arguments to methods for Unix-based systems [15].

```
1 mov    %rbx,%rdi
2 call   <__hook_load8>
3 mov    (%rbx),%r12
```

Listing 1.1. Example assembly code in AT&T style

The body of the hook `__hook_load8()` essentially notes down the instruction-level memory access and uses the address-to-object mapping to translate the

accessed address to the corresponding object metadata. Since it induces prohibitive overhead to track the temporal sequencing of every memory access to every object, Memorizer chooses to keep track of the frequency only.

Also, note that the choice of reusing KASAN means Memorizer loses some memory accesses since KASAN does not add a tracepoint when the access has no chance of incurring memory-safety error.

3.3 Internal Allocator

To avoid stability issues, Memorizer requests a piece of memory from the memblock allocator at boot time and never frees it back to the rest of the kernel. It manages the memory with a simple bump allocator and never frees the memory it has allocated. Therefore, as the system runs, the memory footprint of Memorizer grows monotonically and the kernel will eventually go out of memory and panic.

Hook	Description
`on_start()`	Called when the access monitor is started so that the policy can initialize its own data structure.
`on_exit()`	Called when the monitored process(es) exits so that the policy can free its data structure.
`on_syscall_begin(nr_syscall)`	Called at the beginning of syscalls. `nr_syscall` is the index of the syscall.
`on_syscall_end(nr_syscall)`	Called at the end of syscalls.
`on_mem_access(addr, size)`	Called when machine-level memory access happens. `addr` is the memory address of what's being accessed, and `size` corresponds to the machine instruction.

Table 1. The interface of the monitor framework that is exposed to policy writers. Policies implement those functions and register them with the monitor, which calls them when the corresponding events happen.

4 In-kernel Monitor

In this section, we present our in-kernel monitor framework which targets individual processes. The main purpose of the framework is to allow flexible tracing of low-level features[1], upon which the proposed detector can be quickly prototyped and evaluated.

The layered architecture of the framework is depicted in Figure 2. At the topmost is the policy layer where individual policies process the stream of trace events and decide whether the monitored process is normal or suspicious. One or more policies can be run concurrently in case they are related. Each policy is

[1] A list of the currently supported features can be found in Section 5.1.

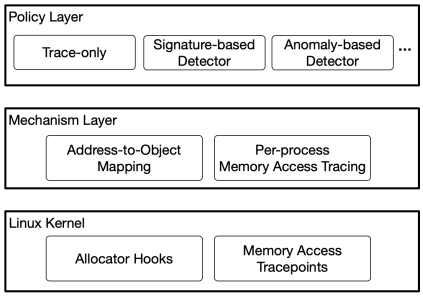


Fig. 2. The layered architecture of the proposed framework.

defined as a set of event handlers, which are registered with the mechanism layer at monitor start time. We explore the design space of some possible policies for exploit detection in Section 5.

Next is the mechanism layer that aims to expose a set of trace events (listed in Table 1) of the monitored process(es) to the policy layer. The key component of the mechanism layer is the efficient per-process tracing framework. At the lowest level, the mechanism is based on the fine-grained tracing techniques pioneered by Memorizer[18] with our specialization and extension to adapt them for our exploit detection use cases. Memorizer is a comprehensive in-kernel tracer that aims to collect memory access traces together with dynamic contexts for the whole system. Like other whole-system tracers, it induces a prohibitively high overhead for the whole system, which makes it unsuitable for online detection. The page table decoupling technique we proposed effectively lowers the whole system overhead.

4.1 Overview

This section presents an end-to-end description of the workflow of the monitor. The monitor can be toggled on and off by a boolean kernel command line parameter. If it is toggled on, at some point in the early-boot stage, the global address-to-object mapping is created and maintained thereafter. Then, right before the `init` process is created, the monitor scans the kernel code for the tracepoints of memory accessing instructions and saves them as a list. The code scanning uses the existing x86 analysis framework in the kernel. After the creation of the list, the tracepoints are toggled off by overwriting them with nops.

The monitor can be started for a process with a custom system call that requires no capability. We build a loader program that makes the call to enable the monitor and use `execve()` to load another binary. When the monitor is turned on for a process, it (1) initializes the process-specific data structures for access tracing (2) duplicates the page tables, and turns on the tracepoints by reverting the nops. This applies to the monitored processes only. Note that, unlike Memorizer, we forgo the kernel-mode activities that have no associated process context, like interrupt handlers. We do not trace page allocation either.

On each system call, the access tracing is toggled on right before the system call handler is invoked. For each tracepoint triggered, the registered memory access handler is invoked. When the system call handler is done, the access tracing is disabled. Finally, when a monitored process quits (i.e., gets terminated), it first switches back to the original page table used by unmonitored processes, then frees the page table of the process.

To enable flexible policies, the monitor exposes a set of hooks for each of the events listed in Table 1. For example, the policy can register an event handler whenever there is memory access. It can also register another handler when the syscall is about to exit. The implementation of the event handler compares the trace of the monitored process with known exploit attempts and decides whether the process is malicious. We present an exploration of different policies in Section 5.

4.2 Per-process Monitoring

Strawman: "if" checking. Unlike Memorizer which targets the whole system, our monitor watches individual processes to mitigate the performance impact. However, the challenge is that all processes share the same piece of code when the system runs in kernel mode. A naive way to enable the monitor for only a subset of processes is to add a runtime check for each of the hook handlers as shown in the code snippet below (Listing 1.2):

```
1 void __hook_load8(unsigned long mem_addr) {
2     if (current->flag && PF_TRUSTED) {
3         return;
4     }
5 }
```

Listing 1.2. A strawman approach for per-process monitoring

Page Table Decoupling. The "if" checking approach above incurs some overhead associated with the branching instructions, as shown in our performance evaluation (Section 6). Instead of runtime checking, we adapt the idea in [12] and let the untrusted process and the trusted processes run slightly different kernels. The trusted processes' tracepoints are nops while the untrusted processes' tracepoints are the actual hooks (i.e. call instructions).

There are many possible ways to run different kernels for different processes. Different from the runtime dispatching mechanism as done in [12], we use different page tables with different kernel text section mappings. The workflow can be described as follow:

1. On boot time, use the in-kernel x86 disassembler to scan the kernel code and build a list of hook addresses.
2. Create fresh copies of all kernel text pages.
3. For the original kernel text pages, disable all hooks by no-oping.

4. When a new monitored process is started, replace its kernel text mapping and point them to the duplicated pages with hooks enabled.

Note that some parts of the kernel assume that the text mapping for all processes is the same. For example, the "static key label" / "jump label" API modifies the text section at runtime to enable/disable certain code paths. In its implementation, it keeps track of the latest state of the associated code address and prints an alert whenever a mismatch is detected. The underlying challenge is to keep the page tables in sync and deal with the differences in the text section. In our implementation, we simply disable the problematic API. A better way to fix this is to hook the existing live patch patching API and remove direct modification with the API call.

4.3 Object-Level Memory Access Tracing

Most of this part is the same as Memorizer, which uses a compiler pass to generate an instruction-level trace and lift the trace to the object level on the fly using the address-to-object mapping described in the previous section (refer to Section 3.2).

While Memorizer hooks the KASAN runtime, to reduce the unnecessary overhead caused by existing KASAN logic, we link in our own runtime for access tracing instead of using the original KASAN runtime.

4.4 Kernel Allocator

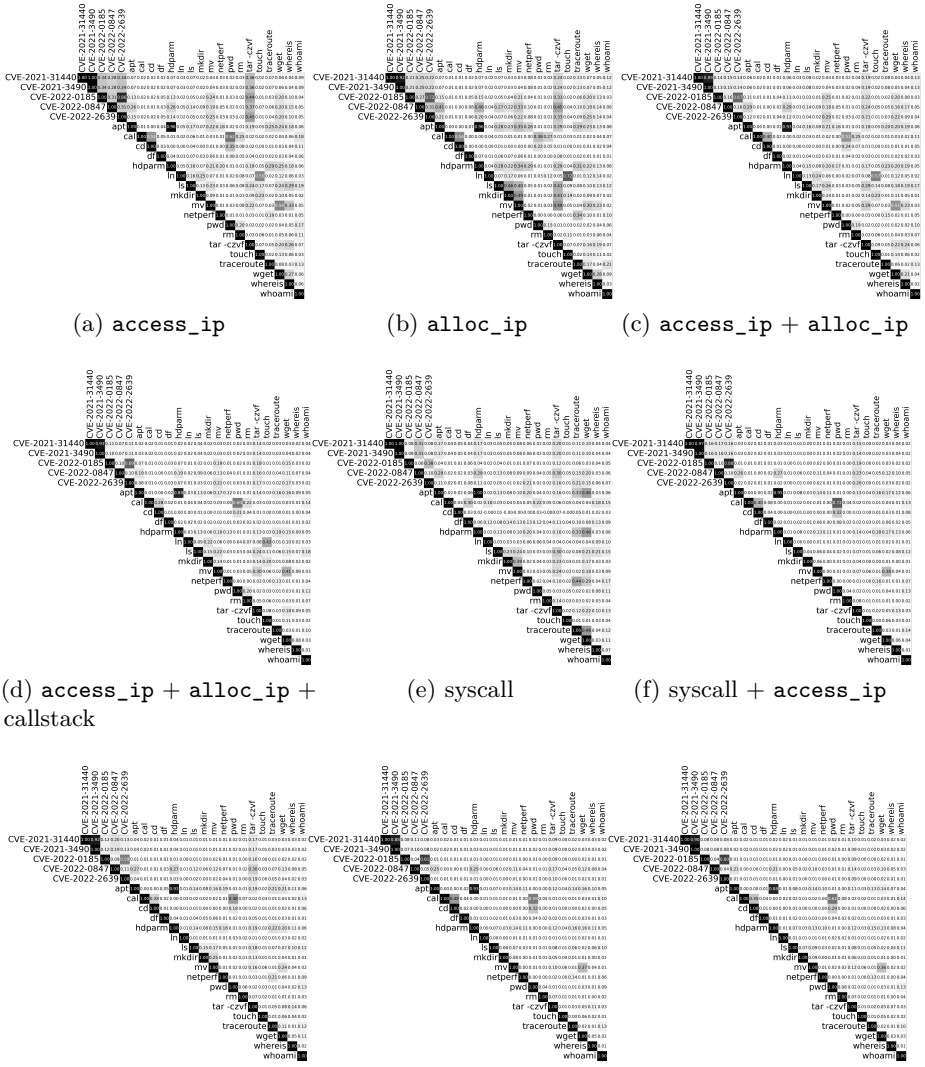
Memorizer reserves a piece of memory from the machine at boot time through the memblock allocator. The bump allocator it uses to manage its own memory can easily cause fragmentation. Instead, we get the memory from the buddy allocator and manage it using a simple slab allocator. This allows better memory reuse and reduces memory overhead. Unlike in Memorizer, the operation of the buddy allocator itself is not tracked.

Another issue is that the kernel allocators are not available at the early-boot stage while the monitor can trace early-boot objects. We delay the initialization of the monitor until the buddy allocator is ready. This means we lose some early-boot objects that Memorizer can track.

5 Policies

As part of the in-kernel monitoring framework, a *policy* is a set of handlers of events listed in Table 1. They can be as simple as a do-nothing policy consisting of empty functions for each handler or as complex as a full-fledged online detector that traces the kernel and kills the monitored process when it detects a malicious behavior. By implementing the detectors as policies we reuse the infrastructure for per-process tracing.

In this section, we study the design space of policies with experiments. For each policy, we first collect data and study the workloads we target by offline



(a) `access_ip` (b) `alloc_ip` (c) `access_ip` + `alloc_ip`

(d) `access_ip` + `alloc_ip` + callstack (e) syscall (f) syscall + `access_ip`

(g) syscall + `alloc_ip` (h) syscall + `access_ip` + `alloc_ip` (i) syscall + `access_ip` + `alloc_ip` + callstack

Fig. 3. Similarity of Normal workloads. This demonstrates the ability a feature can discriminate pairs of workloads. The similarity between a pair of workloads is denoted as a cell in the matrix. The color of the cell indicates the similarity. The brighter the color, the closer the behavior of the two workloads with respect to the feature.

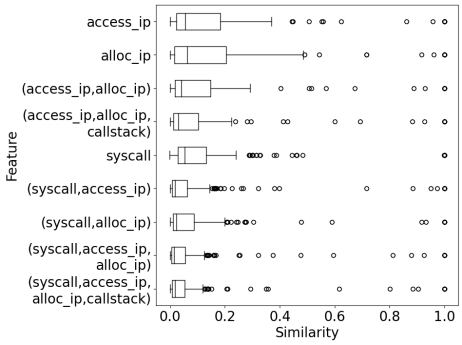


Fig. 4. Distribution of similarity for each feature. This shows the discrimination power of different features.

analysis. Then we present how the policy can be implemented with our online monitor.

In Section 5.1, we begin with a trace-only policy that simply traces various features. Based on the trace-only policy, we describe two detector policies that detect malicious behavior in the following Sections 5.2 and 5.3.

5.1 Trace-Only Policy

The only job of this policy is to record the traces of the kernel-level memory access in terms of some of the following features we are going to study:

- `syscall`: the syscall indices where this memory access happens
- `access_ip`: the instruction address of the memory accessing instruction
- `alloc_ip`: the instruction address of the allocator method (e.g. `kmalloc()`) call of the object being accessed
- `callstack`: the callstack when alloc_ip is executed

The policy maintains a per-process hash table keyed by (a tuple of) the features above. It can be described in terms of the interface exposed by the framework (refer to Table 1) as shown below:

The Trace-Only Policy

- `on_mem_access()`: collect desired features, insert them into the hash table.

5.2 Signature-Based Detector

In this section, we study the feasibility of building a signature-based detector as a policy of our framework. From a high-level view, signature-based detectors define

what is bad first, and anything else is considered good (essentially a "blacklist"). The key is to discriminate the bad from the good. We designed a metric of similarity and evaluate how well they can discriminate exploit attempts from normal workloads. We also experimented with different combinations of features. Finally, we verify our analysis by building a functional prototype.

Data Collection and PreProcessing The data were collected as follows. First, to mitigate indeterminism, we repeat each workload 20 times. Then we take the intersection of each trial of the same workload. Finally, we compare the intersection of each workload with the others. The workloads are shown in the similarity matrices in Figures 3 as tick labels.

Similarity Analysis We adopt the similarity metric of *weighted Jaccard similarity* [9, 19] where the weight is the inverse document frequency (IDF) [10, 17] of the feature.

For a collection of traces $C = \{T_1, T_2, ..., T_n\}$ whose elements can be denoted as $T = \{e_1, e_2, ..., e_n\}$, the IDF of e with respect to the collection C is

$$idf_C(e) = log \frac{|C|}{|\{T \in C : e \in T\}| + 1} \tag{1}$$

where the term in the denominator $|\{T \in C : e \in T\}|$ stands for number of traces that contains e. The more frequent e appears in different traces, the smaller the weight $idf_C(e)$ is. For the sake of simplicity, we shall omit the subscript C in the following text and write $idf(e)$.

For each feature, we take its *inverse document frequency* as the weight. Based on this, we define the *weighted Jaccard similarity* [9] as follows:

$$Jacc_{idf}(T_1, T_2) = \frac{\sum\limits_{e \in T_1 \cap T_2} idf(e)}{\sum\limits_{e \in T_1 \cup T_2} idf(e)} \tag{2}$$

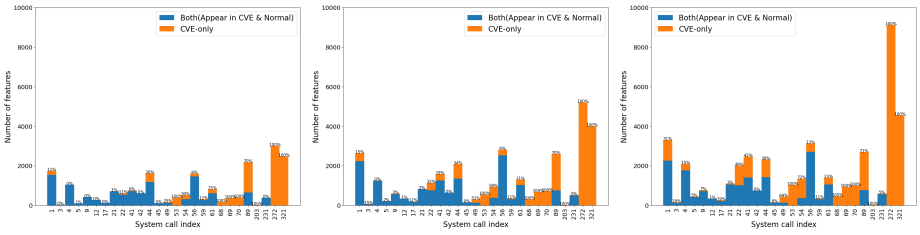
This weighted metric can distinguish the features that are rare in the collection from those that are common. Two traces are likely to be similar if either (1) both of them have little rare features, and the majority of features are shared (2) both of them have many rare features, and lots of rare features are shared.

Figures 3 show the confusion matrices for different features. For each subfigure, both the x-axis and the y-axis range in the same set of workloads. A cell `(x, y)` denotes the similarity of the two corresponding workloads, ranging from 0 to 1. The darker the color, the higher the similarity, and vice versa. Note that the diagonal of the matrix is always 1 (pure black) because the similarity of a workload with itself is always 1. On the other hand, the majority of the cells have a similarity of 0. This means that the detector will be able to capture the differences between the pair of workloads. Some cells have a higher similarity, this can potentially confuse the detector and lead to false positives.

To compare across different features, we summarized each confusion matrix to a violin shown in Figure 4. We can see a general trend that the more features we add, the more likely that the similarity of a pair of different workloads remains low. Based on the analysis above, we conclude that **the signature-based detector is able to discriminate normal workloads against exploit workloads, and adding more features leads to a lower false positive rate generally.** We also observe that system call is a discriminative feature. This motivates the per-syscall analysis in Section 5.3.

	Actually Positive	Actually Negative
Predicted Positive	5	0
Predicted Negative	0	90

Table 2. Confusion matrix of the signature-based detector. The feature is `(syscall, access_ip,alloc_ip)`. The data is collected by running 95 trials with our prototype.



(a) `(syscall, access_ip)` (b) `(syscall, access_ip, alloc_ip)` (c) `(syscall, access_ip, alloc_ip, callstack)`

Fig. 5. Per-syscall anomaly scores for the five exploit workloads for different features. Not all system calls are shown. For example, system calls that are not called in both exploit and workloads or incur too few features are not shown.

Detector Generally, a signature-based detector can be implemented as a policy described below:

	Number of Features
CVE-2022-0185	1135
CVE-2022-0847	813
CVE-2022-2639	1132
CVE-2021-3490	117
CVE-2021-31440	132

Table 3. The size of the signature of different exploits. The feature is `(syscall, access_ip,alloc_ip)`.

The Policy of Signature-Based Detector

- `on_start()`: Read the signature from the file and stores them in a hash table.
- `on_mem_access()`: Collect desired features as keys, and test them against the hash table for signatures. Count the number of distinct matches.
- `on_syscall_exit()`: If the number of distinct feature matches is greater than a threshold, then the workload is considered an exploit.

With the exploit signatures from Section 5.2, we implemented the signature-based detectors for 5 different exploits we collected from the Internet with the target feature `(syscall, access_ip, alloc_ip)`. For each exploit, we drive the monitor with its signature and run both the particular exploit and 18 normal workloads from the previous section. So there are $5 * (1 + 18) = 95$ trials in total. The result is shown in Table 2. Overall, the detector is able to detect all the exploits and no false positives are observed when it monitors normal workloads.

The size of the signature for different exploits is shown in Table 3. The signatures consist of up to 1135 features, which translate to 99,880 bytes with our current prototype implementation. This is negligible compared to the size of the space that other parts of the kernel take.

5.3 Anomaly-Based Detector

In this section, we study the feasibility of building an anomaly-based detector as a policy of our framework. We try different combinations of features shown in Section 5.1 and analyze how well they can discriminate exploit attempts from normal workloads.

From a high-level view, anomaly-based detectors define what is good first, and anything else is considered bad (essentially a "whitelist"). Similar to Section 5.2, the key is to discriminate the bad from the good. We designed yet another[2] similarity metric for this and evaluate how well they can discriminate

[2] A strawman solution is to reuse the definition for the signature-based detector Sec. 5.2 but instead collect the signatures of normal workloads. However, this would not work because (1) Malicious workloads can easily bypass this by doing something normal.

exploit attempts from normal workloads. We also experimented with different combinations of features. Finally, we verify our analysis by building a functional prototype.

Data Collection and PreProcessing We use the same set of workloads (both normal and exploits) in the previous section. Also, we split all the traces by their syscall indices so that we can do per-syscall analysis. For each normal workload, we repeat it 20 times, and 19 of them are considered the "training set" of the anomaly score, while 1 of them remains unused for the "test set". Then we take the union of all the traces in the training set as the known trace. On the other hand, we use two test sets: (1) the separated trials of the normal workloads that are not in the training set as mentioned above (2) trials of all exploit workloads.

Anomaly Analysis Given a new workload, we define the *anomaly score* of the new one N against known ones O as the ratio of features in N that are not in the known trace O:

$$anomaly(N, O) = \begin{cases} \dfrac{|N \setminus O|}{|N|} & \text{if } |N| > 0 \\ 0 & \text{if } |N| = 0 \end{cases} \tag{3}$$

We analyze the data by computing the per-syscall anomaly score of each trial in the test set with regard to the training set. We find that the trials of the normal workloads lead to little anomaly as expected, while the trials of exploits translate into significant anomalies. For the sake of demonstration, we aggregate all exploits and show the results for different features. Refer to Figure 5. The x-axes correspond to system call indices, and the y-axes correspond to the number of features. The orange bars correspond to features that only appear in the exploit workload (test set) but not the normal workload (training set). We can see a number of system calls where the exploit-only features dominate or take a substantial portion. In conclusion, **by focusing on particular system call contexts and watching for anomaly features, we can reliably detect unknown exploits.**

Detector Generally, an anomaly-based detector can be built as a policy described below:

The Policy of Anomaly-Based Detector

- `on_start()`: Read the normal profile from file and stores them in a hash table.
- `on_mem_access()`: Collect desired features as keys, if they haven't appeared before, add them in another hash table for the current workload.
- `on_syscall_exit()`: If the number of unique features in the current workload's hash table is greater than a threshold, then the workload is considered an anomaly.

	Actually Positive	Actually Negative
Predicted Positive	5	0
Predicted Negative	0	18

Table 4. Confusion matrix of the anomaly-based detector. The feature is `(syscall, access_ip, alloc_ip, callstack)`. The data is collected by running 23 trials with our prototype.

	`access_ip`	`alloc_ip`	`callstack`
min	2	2	2
average	207	10	37
max	3852	488	1423

Table 5. The aggregated per-syscall statistics of individual features for the normal profile. The object column corresponds to accessed objects, not the total of the global mapping. This shows the range and the expected number of features in a random syscall used in the normals.

We implement a detector using the features `(syscall, access_ip, alloc_ip , callstack)`. The detector is driven by a single normal profile (described in Sec. 5.3) and can be used to detect any exploit attempts. We test the detector by running the 18 normal workloads which it was built from and the 5 exploits as well. So there are 23 trials in total. The result is shown in Table 4. Overall, the anomaly-based detector can identify all the exploit attempts we have without any false positives, which is consistent with the analysis above.

In terms of memory usage, the normal profile consists of up to 230,919 features, which translates to 21 megabytes with our current prototype implementation. We also show the statistics of unique individual features in Table 5.

6 Performance Evaluation

This section studies the performance of the tracer framework under different features. We begin with the performance of the components of the framework which includes the overhead of maintaining the address-to-object mapping and page table manipulation. Next, we study the performance of different features. We omit the evaluation of the two detector policies because they add negligible overhead based on the trace-only policy by design. Finally, we present the memory usage of the framework.

Our kernel is implemented based on mainline Linux kernel v5.10. The performance evaluation was done on bare metal equipped with AMD Ryzen 9 3900X and 128G memory. The OS and compiler we use are Ubuntu 20.04 and `gcc9.4.0` respectively. The kernel configuration file `.config` is taken from the Ubuntu kernel with our customization and the jump label API disabled (see Section 4.2).

Process Times (microseconds)					
	null call	null I/O	stat	open/clos	slct TCP
vanilla Linux	0.05	0.09	0.36	0.81	3.07
pgtbl.	0.05	0.1	0.45	0.91	3.93
"if" check	0.14	0.61	3.83	5.96	39
obj. tracing + pgtbl.	0.14	0.61	4.1	7.24	37.5
obj. tracing + "if" check	0.14	0.62	4.08	7.19	37.3
	sig inst	sig hndl	fork proc	exec proc	sh proc
vanilla Linux	0.1	0.54	104	189	722
pgtbl.	0.1	0.6	110	231	821
"if" check	0.25	2.85	240	573	1664
obj. tracing + pgtbl.	0.26	3.06	291	853	2105
obj. tracing + "if" check	0.25	293	521	1863	
Local Communication Bandwith (MB/s)					
	Pipe	AF UNIX	TCP	File reread	Mmap reread
vanilla Linux	5040	10000	8025	12500	18200
pgtbl.	4170	10000	7582	11400	18200
"if" check	1677	6241	1808	4702.9	18100
obj. tracing + pgtbl.	1689	5290	2080	4634.3	17900
obj. tracing + "if" check	1581	5245	1708	4665.9	18100
	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write	
vanilla Linux	19600	10300	16000	15500	
pgtbl.	20000	10300	16000	16300	
"if" check	19600	10000	16000	15800	
obj. tracing + pgtbl.	19500	10400	16000	16200	
obj. tracing + "if" check	19600	10200	16000	15500	

Table 6. LMbench Evaluation for trusted processes. Feature is `(access_ip, alloc_ip)`

	apache (reqs/sec)	postgresql (TPS)	openssl (signs/sec)	compress-7zip (MIPS)	tensorflow (images/sec)
Trusted Processes					
vanilla Linux	35687.38	493	193	103240	12.29
pgtbl.	31305.83	489	185.9	97683	12.03
"if" check	6268.55	387	189.3	101994	12
obj. tracing + pgtbl.	5339.64	381	185.7	101919	11.96
obj. tracing + "if" check	5385.6	379	185.2	101590	12
Untrusted Processes					
access_ip	1653.55	202	190.8	100407	10.69
alloc_ip	1185.62	186	188.9	100427	10.39
access_ip+alloc_ip	1135.13	184	186.9	99100	10.4
access_ip+alloc_ip+callstack	1022.5	183	191.1	98974	10.37

Table 7. Phoronix Test Suite Evaluation for trusted and untrusted processes. For untrusted processes, page table decoupling is tured on. Object tracing is also turned on except for the `access_ip` case.

First of all, observe that our monitor does not decrease the pure memory IO performance in any cases (Tables 6, 7): the performance of the Bcopy[3], Mem read/write in lmbench and the memory-intensive macro test cases (openssl, comrpress-7zip, tensorflow) stays almost the same.

Trusted Processes To begin with, observe that the proposed page table manipulation mechanism for no-oping tracepoints can save us a lot of overhead compared to just guarding them with branch checking. To see this, first compare the vanilla Linux (row 1) with the "if" checking way (row 3) in Tables 6, 7. We can see that the tracepoints significantly decrease the performance of the system in almost all metrics. On the other hand, the proposed page table manipulation mechanism (row 2) yields performance comparable to vanilla Linux.

However, when object tracing is enabled (rows 4, 5 in Tables 6, 7), the performance decreases significantly. That is because the monitor maintains a per-byte mapping to shadow objects with an interval tree, which requires $O(logn)$ insertion time where n is the object size. The performance burden overshadows the benefit brought by no-oping tracepoints. We discuss potential ways to address this in Section **??**.

Untrusted Processes The performance overhead of untrusted (i.e. monitored) processes is shown in Table 7. Again, for memory-intensive workloads, the performance barely decreases, while for network-intensive and disk-intensive workloads the performance decreases significantly (up to 98% degradation). This is because we monitor the kernel activity only, and disk and network IO is generally syscall-dense.

Different Features Table 7 shows the overhead of different features for untrusted processes. The general trend is that the more features we use, the more overhead we have. This is because as the type of features increases, the number of unique features is likely to increase.

Memory Consumption After booting the system, the framework uses a total of 78MB of memory for the global address-to-object mapping. After running lmbench with the trace-only policy, new objects take an additional 638MB of memory. On the other hand, 112,993 unique features are recorded, which takes a total of ∼10MB of memory. Our current implementation uses a hash table of 32MB to store those features, and the size of the table can be tweaked easily. The memory usage of the features is negligible as we are tracing unique types of features. For the detector-based policies, an additional hash table of similar size is needed to store the signatures/normal profile, but their size is again negligible (up to a few tens of megabytes), as mentioned in Sections 5.2 and 5.3.

[3] `Bcopy()` is similar to `memcpy()`. It was depreciated in POSIX.1-2001 and removed in POSIX.1-2008.

7 Discussion and Future Work

We hope to supplement them with the following in future efforts.

Practical Tool While our aim in this work was to examine the potential for context-sensitive memory access patterns to distinguish exploits, an efficient industry level state of the art intrusion detection system requires significant enhancesments. Our future work includes the addition of an optimization algorithm that uses an enriched data set of normal and exploit behaviors to hit a performance budget of observed memory access to likelyhood of attack detection. Given the results already we believe a reduced fingerprint based on the most different but least frequently accessed objects will provide a powerful monitor while still providing enough coverage to detect evasion.

Enriched Signature Analysis We run Hi-Res on different classes and create kernel level memory access footprints of each type, so that we can create a classifier for different CWEs, and further the ability to detect anomalous behavior. Furthermore, we aim to model and examine known rootkits to provide effective working set models to identify common exploit fingerprints in the grammar of the working sets.

Expanding Dyanimic Contexts and Guided Fuzzing Kernel level memory accesses describe program behavior and provide a context sensitive view; Contexts obtained from Hi-Res reduce the reachability of the probable program execution states, potentially providing ways to enhanced fuzzing with Hi-Res based mutation fuzzing.

Vulnerability Research and Reverse Engineering Since our approach only requires instrumenting the kernel, this tool can also be used to improve reverse engineering endeavours.

Formalizing the Model The present description is a bottom up real implementation of a powerful set of concepts that interrelates representations in the dynamic runtime. This effectively operates as a dynamic subtyping of program execution. We seek to formalize the interface to Hi-Res and make available rich property based analysis with rigorous form methods and verification features. One such endeavour will be to describe Hi-Res with a language and formalize the workings sets as type system theory.

8 Conclusion

In this paper, we present an in-kernel exploit detector framework based on fine-grained kernel behaviors. Our evaluation shows that the detectors can effectively discriminate exploit attempts with acceptable overhead. We further evaluate the detectors with different types of features and conclude that system call is the

most effective feature while adding more features results in marginal benefits. We believe that the Hi-Res and the exploration we conducted lay a foundation for future data-driven kernel hardening mechanisms based on richer fine-grained kernel behaviors and more sophisticated policies.

References

1. Kernel address sanitizer (kasan) (Dec 2022), URL https://web.archive.org/web/20221224061314/https://google.github.io/kernel-sanitizers/KASAN
2. Linux-kernel-defence-map: Linux kernel defence map shows the relationships between vulnerability classes, exploitation techniques, bug detection mechanisms, and defence technologies (Oct 2022), URL https://web.archive.org/web/20230109052513/http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project
3. a13xp0p0v: Kernel self protection project (Aug 2022), URL https://web.archive.org/web/20221211193519/https://github.com/a13xp0p0v/linux-kernel-defence-map
4. Cook, K.: Lwn.net: Kernel self protection project (Nov 2015), URL https://web.archive.org/web/20221220112558/https://lwn.net/Articles/663361/
5. Cowan, C., Beattie, S., Day, R.F., Pu, C., Wagle, P., Walthinsen, E.: Protecting systems from stack smashing attacks with stackguard. In: Linux Expo (1999)
6. Edge, J.: State of the kernel self protection project (Aug 2016), URL https://web.archive.org/web/20221006073412/https://lwn.net/Articles/698827/
7. Ezeme, O.M., Mahmoud, Q.H., Azim, A.: Dream: deep recursive attentive model for anomaly detection in kernel events. IEEE Access **7**, 18860–18870 (2019)
8. Hou, S., Saas, A., Chen, L., Ye, Y.: Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In: 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW), pp. 104–111, IEEE (2016)
9. Ioffe, S.: Improved consistent sampling, weighted minhash and l1 sketching. In: 2010 IEEE international conference on data mining, pp. 246–255, IEEE (2010)
10. Jones, K.S.: A statistical interpretation of term specificity and its application in retrieval. Journal of documentation (1972)
11. Kemerlis, V.P., Polychronakis, M., Keromytis, A.D.: ret2dir: Rethinking kernel isolation. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14), pp. 957–972 (2014)
12. Kurmus, A., Zippel, R.: A tale of two kernels: Towards ending kernel hardening wars with split kernel. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1366–1377 (2014)
13. Lampson, B.W.: Protection. ACM SIGOPS Operating Systems Review **8**(1), 18–24 (1974)
14. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., et al.: Meltdown: Reading kernel memory from user space. Communications of the ACM **63**(6), 46–56 (2020)
15. Matz, M., Hubicka, J., Jaeger, A., Mitchell, M.: System v application binary interface. AMD64 Architecture Processor Supplement, Draft v0 **99**(2013), 57 (2013)
16. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. In: Pioneers and their contributions to software engineering, pp. 479–498, Springer (1972)
17. Robertson, S.: Understanding inverse document frequency: on theoretical arguments for idf. Journal of documentation (2004)

18. Roessler, N., Chien, Y., Atayde, L., Yang, P., Palmer, I., Gray, L., Dautenhahn, N.: Lossless instruction-to-object memory tracing in the linux kernel. In: Proceedings of the 14th ACM International Conference on Systems and Storage, pp. 1–12 (2021)
19. Shrivastava, A.: Simple and efficient weighted minwise hashing. Advances in Neural Information Processing Systems **29** (2016)
20. Silberman, P., Johnson, R.: A comparison of buffer overflow prevention implementations and weaknesses. IDEFENSE, August (2004)
21. Sun, J., Zhou, X., Shen, W., Zhou, Y., Ren, K.: Pesc: A per system-call stack canary design for linux kernel. In: Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy, pp. 365–375 (2020)
22. Wang, X., Li, C.: Android malware detection through machine learning on kernel task structures. Neurocomputing **435**, 126–150 (2021)
23. Xie, W., Xu, S., Zou, S., Xi, J.: A system-call behavior language system for malware detection using a sensitivity-based lstm model. In: Proceedings of the 3rd International Conference on Computer Science and Software Engineering, pp. 112–118 (2020)
24. Zhang, X., Mathur, A., Zhao, L., Rahmat, S., Niyaz, Q., Javaid, A., Yang, X.: An early detection of android malware using system calls based machine learning model. In: Proceedings of the 17th International Conference on Availability, Reliability and Security, pp. 1–9 (2022)